



TECHNICAL MEMORANDUM

**Department of
Computer Science**



A DISTRIBUTED IMPLEMENTATION METHOD
FOR PARALLEL PROGRAMMING

Alain J. Martin
California Institute of Technology
Pasadena, California 91125

5045:TM:80

Appears in INFORMATION PROCESSING 80, S.H. Lavington(ed.)
North-Holland Publishing Company

A DISTRIBUTED IMPLEMENTATION METHOD FOR PARALLEL PROGRAMMING

A.J. MARTIN

Philips Research Laboratories

5656 AA Eindhoven, The Netherlands

A method is described for implementing on a finite network of processing "cells", called the "implementation graph", programs whose potential parallelism is not fixed by the implementation but varies according to the input parameters. First, programming constructs are described permitting a computation, regarded as a dynamic structure called the "computation graph", to diffuse through the implementation graph. Second, the implementation problem of mapping an unbounded number of computation nodes on a finite number of cells is tackled. Processor allocation and message buffering completely disappear from the programmer's concerns. The mechanism proposed is considered a generalization of the stack mechanism.

1. INTRODUCTION

A program is built from elementary actions and control structures. For a sequential program, the control structures are such that any computation performed by an automaton under control of the program is a sequence of elementary actions totally ordered in time. We call "parallel program" a program in which some control structures allow elementary or composite actions to be performed simultaneously. A "distributed automaton" is a finite network of sequential automata, each with its own store and processor - we call such a sequential automaton a "cell" - , where the communications between cells are restricted: each cell communicates with only a proper subset of the other cells (its neighbours). There is no common store and no general communication network.

A method is proposed for designing highly parallel programs, and for implementing them on distributed automata. Its guidelines are the following. The potential parallelism of a program is not fixed beforehand, but varies according to the input parameters as do the depth of a recursive computation and the number of steps of an iteration (this approach has been suggested by C.A.R. Hoare in [1]). The parallel component actions of a computation - called "nodes" - are created (and destroyed) as the computation proceeds. They are regarded as the vertices of a graph - called the "computation graph" - , which grows and shrinks according to the needs of the computation. A directed edge from node A to node B indicates that A has created B, and that A and B may communicate with each other. Such a graph represents a partial order of the nodes, and all nodes that are not ordered according to this relation are called parallel nodes.

The unbounded freedom of creating nodes has, however, to be reconciled with the finiteness and the topological restrictions of a distributed automaton. Such an automaton is represented by a (now fixed and finite) graph - called the "implementation graph" - whose vertices are the cells, and whose edges represent the communication possibilities between cells. In fact, the computation graph formalizes the topological needs of a computation,

and the implementation graph the topological constraints imposed on the computation by the automaton: the (possibly unbounded) number of variables and of parallel operations on these variables must be distributed over a finite number of processing and storage elements - the cells - with limited communication means among them. In other words, an a priori unbounded computation graph must be mapped on a finite implementation graph.

This mapping poses two major problems. Firstly, the complexity of communication actions (and thus of the whole computation) can become unpredictable if two neighbour nodes are mapped on two non-neighbour cells. Secondly, the sharing of a fixed number of processors by an unbounded number of communicating activities can introduce deadlock. As will be shown, these two problems will be solved by the single "neighbourhood requirement":

two neighbour nodes of the computation graph are mapped on two neighbour cells of the implementation graph.

We shall first introduce the principal programming constructs allowing a computation graph to grow and shrink "through" an implementation graph. We shall then describe the implementation techniques for mapping an unbounded number of nodes on a finite number of cells.

2. THE PROGRAMMING CONSTRUCTS

2.1 An example

Consider the recursive computation of the combination function, or binomial coefficient $C(n,k)$:

$$C(n,k) = \begin{cases} \text{if } k < 0 \vee n < k \rightarrow 0 \\ 0 = k \vee k = n \rightarrow 1 \\ 0 < k \wedge k < n \rightarrow \\ C(n-1,k) + C(n-1,k-1) \end{cases}$$

fi.

The computation of $C(n,k)$ for $0 < k < n$ requires the computation of two new coefficients:

$C(n-1,k)$ and $C(n-1,k-1)$. Obviously, the computations of these two coefficients may be performed in an arbitrary order, and since they do not share variables, they may even be performed simultaneously - in parallel - without further precaution.

For instance, the computation of $C(4,2)$ generates the following tree of procedure activations and evaluations:

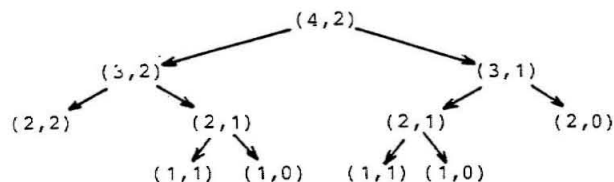


Fig. 1.

This tree is a possible computation graph for the computation of $C(4,2)$. But it should be possible to design a program generating the following computation graph for $C(4,2)$:

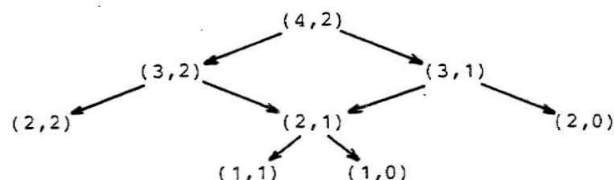


Fig. 2.

This solution presents the advantage that each coefficient is evaluated only once. For the sake of clarity, we shall confine the description of the method to programs generating computation trees. The problems of other computation graphs will be discussed later.

2.2 Nodes and the procedure mechanism

In the computation of the binomial coefficient, parallelism is introduced by calling the procedures $C(n-1,k)$ and $C(n-1,k-1)$ in parallel. In this example, each procedure call creates a new "procedure instance". Such a procedure instance corresponds to a node of the computation graph and is therefore called a node.

Hence, a node consists of a set of local variables and a program text with its own instruction counter. The local variables of a node that are used to communicate with other nodes are called "parameters". An edge of the computation graph is called a "channel". Each node possesses a (non-empty) set of channel names. A channel name uniquely identifies a channel incident to the node. A node may also possess channel variables to which channel names are assigned.

Assume that a node N_1 performs a procedure call with the net effect of assigning the value of $C(n,k)$ to a variable r .

- (1) A new node N_2 is created: N_1 is the father of N_2 and N_2 the son of N_1 , and N_1 and N_2 are linked to each other by a channel, say H .
- (2) The value of the parameter (n,k) is sent from N_1 to N_2 , over H .
- (3) The value of $C(n,k)$ is computed by N_2 , and sent from N_2 to N_1 , over H . As a result, the

value of $C(n,k)$ is assigned to r .

In the procedure call, sending and receiving parameter values are two explicit communication actions. In the procedure body, the complementary actions of receiving and sending parameter values also appear as explicit communication actions.

2.3 The parallel program for $C(n,k)$

Before introducing the different programming constructs needed, we give, as an example of their use, the text of the procedure for the parallel computation of $C(n,k)$. (Since it is definitely not the purpose of this paper to introduce a new programming language, the syntax used will remain largely undefined. But it should be quite clear for anyone familiar with modern programming languages.)

```

1  proc C(F?(n,k), F!r)
2      F: channel; n,k,r: integer;
3  begin
4      F?(n,k);
5      if k < 0 or n < k then r := 0
6      or 0 = k or k = n then r := 1
7      or 0 < k and k < n then
8          begin S1,S2: channel; r1,r2: integer;
9              [C(S1!(n-1,k); S1?r1)
10               // C(S2!(n-1,k-1); S2?r2)];
11              r := r1 + r2
12          end
13  fi;
14  F!r
15  end

```

prog. 1.

2.4 The communication mechanism

The communication mechanism used is similar to the one proposed by C.A.R. Hoare in [1]. The transmission of the value (of the expression) x from N_1 to N_2 is the coincidence of the send action $S!x$ in N_1 and the receive action $F?y$ in N_2 : S is the name of the channel (N_1,N_2) in N_1 , F is the name of the channel (N_1,N_2) in N_2 , and y is a local variable of N_2 . As a result of the communication action, the value (of the expression) x has been assigned to y .

A started communication action (send or receive) on a channel is delayed (we say it is "pending") until the complementary communication action (receive or send) on the same channel is also pending. The two actions are then simultaneously completed: we say that the communication "fires".

A special form of receive action, called "selection" is defined, involving a channel variable that has previously been declared together with a set of possible channel names: for instance, the channel variable V can be declared of the type $\{left, right\}$ where "left" and "right" are the names used in this node for two channels. If a send action is pending in another node on one of these two channels, this send action is selected as the complementary

action of the selection, and the communication fires on the selected channel. As a result, the local name of the selected channel is assigned to V over the whole scope of the declaration of V .

2.5 The procedure call

Given the syntax of send and receive actions, and given that the transmission of parameters are explicit communication actions in the procedure call and in the procedure body, the syntax used in prog. 1 for the procedure call and heading is now quite clear. Prog. 1 contains two procedure calls:

```
C(S1!(n-1,k); S1?r1)   at line 9,
C(S2!(n-1,k-1); S2?r2) at line 10,
```

combined in a control structure - the "parallel construct" - that we shall describe later. The procedure call of line 9 consists of two distinct communication actions (to be performed in this order, hence the semicolon between them):

- 1) the send action $S1!(n-1,k)$,
- 2) the receive action $S1?r1$,

and similarly for the procedure call of line 10. The complementary actions can be found in the procedure body, namely the receive action $F?(n,k)$ of line 4 and the send action $F!r$ of line 14. In the procedure heading, - lines 1 and 2 - the communication actions with the father node are repeated in the form in which they occur in the body.

2.6 Channel declaration

The procedure heading of a procedure declaration - lines 1 and 2 of prog. 1 - defines the interface of a node with the "outside world", i.e. with its father. Only channel F , which links the node to its father, need be declared in the procedure heading: such a channel is called an external channel. The channels $S1$ and $S2$, declared at line 8, are internal channels: they are not known from the "outside world".

Upon declaration of an internal channel name -- for instance S in N -- a new node is created, which is identified in N by S .

In prog. 1, upon declaration of $S1$ and $S2$ at line 8, not only are the two names $S1$ and $S2$ introduced, but two new nodes $N1$ and $N2$ are created related to the father node by $S1$ and $S2$, respectively. Hence, each channel, which relates a father and a son node, has two names: the external name F in the son node, and the internal name $S1$ or $S2$ in the father node. Unlike the usual procedure mechanism, which creates a new anonymous procedure instance at each procedure call, a node is created only upon declaration of an internal channel. This permits a father node to communicate several times with the same son node.

The channel mechanism described above permits a computation graph to grow and shrink (a node disappears after the completion of the last statement of its procedure body) during a computation.

2.7 The parallel construct

The control structure $[...//...]$ of lines 9 and 10 is called a "parallel construct". Its semantics is the following.

Let $P1, P2, \dots, Pn$ be n program parts. The net effect of the parallel composition $[P1//P2//\dots//Pn]$ of these program parts is equivalent to the net effect of any sequence Q of program parts:

$A1; A2; \dots; Am$ ($m \geq n$)

where:

- . the Ai 's are indivisible actions composing the Pj 's,
- . all indivisible actions composing a Pj are present in Q ,
- . if Ai and Ak belong to the same Pj , they must occur in Q in the same order as in Pj ,
- . an indivisible action is either a communication action (until its suspension or its completion) or the sequential program part between two consecutive communication actions. (We have chosen the coarsest "grain of interleaving".)

2.8 The mapping of channels on links

The programming constructs presented so far do not enable us to map a computation graph on an implementation graph (in this respect, prog. 1 is incomplete). This mapping will be achieved by associating with each channel an edge of the implementation graph. An edge of the implementation graph is called a link. Link names are local. In order to allow a program to communicate with the environment, at least one cell must be provided with an external link, i.e. a link connected to no other cell of the graph. Such a cell is called a root. (Links which are not external are, of course, internal.)

Upon declaration, a channel name must be given a certain link-type.

Let $C1$ and $C2$ be two cells linked to each other by a link called L in $C1$. Let $N1$ be a node "located in $C1$ ". If, inside $N1$ an internal channel name S is declared of link-type L , the net effect of the declaration is to create a new node $N2$ located in $C2$. This is the only way to locate nodes in cells.

The first procedure call (from the "outside world") will necessarily locate the root node of the computation tree in one root cell.

Assume that we want to distribute the computations generated by prog. 1 over the implementation graph G of fig. 3.

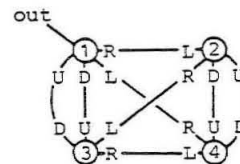


Fig. 3.

G comprises four cells numbered from 1 through 4. The link "out" is the only external link, and thus cell 1 is the only root. Each cell has four internal links incident to it, named U , D ,

R, L. Let us replace line 8 of prog. 1, by the declaration:

"begin S1: channel on D, S2: channel on R;"

and let us assume that the node (4,2) -- i.e. the node computing C(4,2) according to prog. 1 -- has been created in cell 1 by a procedure call "from the outside world". When the declaration of line 8 is reached, two son nodes of (4,2) are created: one in cell 2 by the declaration of S2, one in cell 3 by the declaration of S1. According to the procedure calls of lines 9 and 10, the node created in cell 2 is (3,1), and the node created in cell 3 is (3,2). By the same process, (3,1) will create the son nodes (2,1) and (2,0) in cell 4 and cell 3, respectively; (3,2) will create the son nodes (2,2) and (2,1) in cell 1 and cell 4, respectively; etc...

We see that the external channel name F is not always of the same link-type: in node (4,2) it is of type "out", in node (3,1) of type "L", in node (3,2) of type "U", etc... In such cases, a set - say, FL - of possible father-links is declared in each cell:

FL={U, L, out} in cell 1,

FL={U, L} in cells 2, 3, 4,

and the external channel F is declared in the heading:

F: channel on FL.

It is upon creation of the node that the selected link is associated to the channel name.

Obviously, the mappings of computation graphs on implementation graphs according to the procedure described fulfils the neighbourhood requirement.

* *
*

Since our initial objective was the distribution of an unbounded number of parallel activities (the nodes) over a fixed finite number of cells, at least one cell must accommodate an unbounded number of nodes. As we strive for a homogeneous spreading of nodes over cells, in fact each cell will have to accommodate an unbounded number of nodes. In such a case, the implementation of nodes on a cell, considered as an automaton with finite capacities in processing, storage, and communication media, poses a number of serious problems.

3. THE INTERLEAVING OF NODES IN A CELL

If we consider a cell as a sequential automaton -- or process --, the problem is to interleave the constituent actions of the different nodes of a cell in a sequence of totally ordered actions. The main difficulty is to guarantee that the total ordering introduced does not create deadlock. A deadlock occurs as soon as two actions, which should be ordered in a particular way because they belong to two nodes on the same computation path, are ordered in a different way in the cell. (A computation path is an ordered path of the computation graph.)

We shall propose a deadlock-free strategy for interleaving the nodes in a cell, in which the obligatory order relation between some nodes of the cell need not be known. In fact, we shall interleave the nodes as the component actions of a parallel construct such as defined in 2.7. When a node has been selected as the next step in the activity of a cell, it proceeds from one communication action to the next one, or terminates. Assume that such a next step has to be chosen. All nodes have reached one pending communication action. The set of pending communication actions inside a cell is called the pending set. As soon as one communication action becomes firable (the complementary communication action is pending in a neighbour cell), the node it belongs to is chosen as the next step of the cell activity. When several communication actions are firable, an arbitrary one is chosen.

Theorem: If the mapping of nodes on cells fulfils the neighbourhood requirement, the above strategy for interleaving nodes in a cell is deadlock-free.

Proof: Assume that:

- (1) The computation is deadlock-free in absence of an implementation graph (implementation-free computation).

We shall prove that the same computation mapped on an implementation graph according to the above strategy (the "implemented computation") is still deadlock-free, i.e. at any time at least one communication can fire or the computation has terminated.

The strategy introduces restrictions on the computation parallelism: parallel nodes mapped on the same cell can no longer proceed in parallel (they are interleaved according to the interleaving rule of the parallel construct). This restricts the class of possible states reached by an implemented computation but the class of states reached is still a subset of the class of states reached by the implementation-free computation. Hence the lemma:

- (2) The union of all pending sets, - i.e. taken over all cells - is a possible set of pending communications reached at some stage by the implementation-free computation.

From (1) and (2):

Either the computation has terminated or at least one pending set contains a send action - say, in node N of cell C - the matching receive action of which is also pending - either in the father node F or in a son node S of N - . (Communications along an external link, i.e. with the environment, are assumed always firable.) But according to the neighbourhood requirement, F and S are mapped on two cells which are both neighbour cells of C. Hence, according to the above strategy, the send action of N and the receive action of F or S can fire. (End of proof)

4. THE HANDSHAKING PROTOCOL

In the previous section, it has been said that a communication action of a pending set is

firable if and only if the complementary action is pending. Handshaking is the protocol by which two complementary firable communication actions are selected to fire. The main difficulty is to avoid mismatch. A mismatch is the situation where node N1 decides to communicate with node N2, and N2 decides to communicate with a third node N3. To each node, two sets P and Q of booleans are attached: in each set, one boolean per channel. For a given node N, let $N.P[X]$ and $N.Q[X]$ be the two booleans attached to the channel X. Our purpose is to prove that (at relevant places):

$$\neg N.P[X] \vee \{ \text{a communication on X} \\ \text{is pending in N} \} \quad (1)$$

$$\neg N.Q[X] \vee \{ \text{a communication on X} \\ \text{is pending in N'} \} \quad (2)$$

where N' is the node sharing X with N. Initially, $\neg N.P[X]$ and $\neg N.Q[X]$ hold for all X. The protocol relies on the existence of an elementary communication action called signalling: a signal from N' on X sets $N.Q[X]$ true, a signal from N on X sets $N'.Q[X]$ true. The SEND and SELECT procedures can be described as follows (the RECEIVE procedure is implemented as a SELECT on a set containing exactly one element.)

SEND (on channel X in node N):

```
N.P[X] := true;
N'.Q[X] := true;
wait N.Q[X];
"FIRE SEND"; N.Q[X] := false;
N.P[X] := false.
```

SELECT (on a set S of channels, in node N'):

```
A X in S: N'.P[X] := true;
wait (E A in S: N'.Q[A]);
N'.Q[A] := false; N.Q[A] := true;
"FIRE SELECT";
A X in S: N'.P[X] := false.
```

The assignments $N'.Q[X] := \text{true}$ in SEND, and $N.Q[A] := \text{true}$ in SELECT are signals. This way of describing signals expresses that they are unsynchronized communication actions: a signal performed by N terminates independently of the state of N'.

The statement "wait B" terminates if and only if B holds. In the implementation of communication actions, the wait statement is the place where switching from one node to another occurs: when the current node reaches a wait for which B does not hold, the node is suspended and the pending set is scanned until another firable communication is found, i.e. a node M for which $(E Y: M.P[Y] \wedge M.Q[Y])$ holds. The node M is then chosen as the next step in the activity of the cell. Observe that this is the only place where the set P and the invariant (1) are needed. It is easy to verify that (1) holds for all suspended nodes. We shall only sketch the structure of the proof; the details are left to the reader.

I) It is easy to verify that (2) holds for all nodes. Hence, since the firing condition of a SEND is $N.Q[X]$, and the firing condition of a SELECT is $(E A \text{ in } S: N.Q[A])$, these firing conditions conform to the chosen semantics of communication actions.

II) The impossibility of mismatch is a direct consequence of the fact that a SELECT sends exactly one signal.

III) The situation cannot occur where a node N is suspended in a SEND on channel X, and a node N' is suspended in a SELECT on a set S comprising the same channel X.

For N', would hold: $(A Y \text{ in } S: \neg N'.Q[Y])$.

For N, would hold: $N'.Q[X]$.

A contradiction.

In the case of terminating programs, this is sufficient to guarantee that any firable communication eventually fires.

5. CHANNEL IMPLEMENTATION AND STACK MECHANISM

For the nodes of a same cell, the primitive communication actions (FIRE SEND, FIRE SELECT, signal) on all channels of a certain link-type are "simulated" by equivalent communication actions on the unique link. An extra parameter, called the "label", is transmitted for identifying the channel. If nothing is specified, the internal name of the channel is considered local, and an implicit renaming procedure (which we shall not describe) transforms the internal name in such a way that no two channels have the same name. The transformed name is used as label. This is the most usual case. But it restricts the computation graphs to trees since it is then impossible for one node to be referred to by more than one father.

If an internal channel name is explicitly declared global, no renaming takes place: the internal name is directly used as label. This makes it possible for several internal channels of different father nodes to be given the same name. And thus, this makes it possible for a node to have several fathers; this is the way to generate computation graphs other than trees.

The label is associated to the external name of a channel upon creation of the node. The general method to create a node is to use a "declaration message". If we assume that a node procedure always contains argument parameters, the creation of a node may occur upon receipt of the first signal with a given label.

The creation of a node is very similar to the creation of a procedure instance in the usual stack mechanism. Unlike the usual stack mechanism, the termination of nodes does not take place in a last-in-first-out order, since the nodes of a cell are in general not ordered. It is important to observe that, since parameter variables are allocated on the stack upon node creation, as normal local variables of a node, no explicit buffering is necessary for messages. The similitude with the stack mechanism goes even further. Consider the implementation graph consisting of one cell with self-loops as links. The mechanism described then reduces to a non-deterministic form of the sequential stack mechanism. Hence the mechanism described can

be regarded as a consistent generalization of the sequential stack mechanism.

6. A LAST EXAMPLE

Up till now, nodes communicated only via the parameter mechanism: all variables were local. But it is possible to let the nodes of a same cell share variables: such variables are called "cell variables". A reason for tackling another example is to show how cell variables can be used. (In connection with the sharing of variables between parallel nodes, it is worth remembering that, thanks to the interleaving strategy, the program parts between two communication actions are indivisible actions.)

The problem is to find the length of the shortest path from a vertex, the source, to each other vertex of a strongly connected directed graph G . G is used as implementation graph: each cell is identified with a vertex of the graph, and each link of the cell with an edge of the vertex. Two disjoint sets of links are distinguished: the set $X = \{X_0, X_1, \dots, X_r\}$ corresponding to incoming edges, the set $Y = \{Y_0, Y_1, \dots, Y_q\}$ corresponding to outgoing edges. Each cell contains a cell variable l recording the current shortest length to that vertex. The source is considered a root cell of the graph. The outside world creates a node in the source by the program:

```
begin p: integer; C: channel;
p:= 0; P(C!p; C?())
end
```

where $()$ is an empty parameter.

The problem is to design the procedure P such that

- I) all created nodes terminate,
- II) when they have terminated, in each cell, l is equal to the length of the shortest path from the source to that cell.

The solution is as follows.

Initially l is equal to $+\infty$.

Each cell also contains a set of constants:

e_0, e_1, \dots, e_q equal to the lengths of the outgoing edges Y_0, Y_1, \dots, Y_q , respectively.

```
proc P(F?p, F!()) F: channel on X;
p: integer;

begin
F?p;
if p  $\geq$  1 - skip
  || p < 1 -
    l:= p;
    begin A i:0..q: Si: channel on Yi;
      [// i:0..q: P(Si!(p+ei); Si?())]
    end
  fi;
F!()
end.
```

Prog. 2.

(The proof is left to the reader.)

We have a second reason for showing this example. Consider prog. 2 without the transmission of empty results. It also computes the lengths of the shortest paths. The only difference is that the outside world cannot detect the termination of the computation. Hence, in this example, the whole stacking mechanism is only used for the detection of termination. This method for termination detection shows strong similarities with the one proposed by E.W. Dijkstra and C.S. Scholten in [2].

7. CONCLUSION

The method enforces a clear separation of concerns between, on the one hand, the design of programs independently of any sequential or parallel computational model, and on the other hand, their implementations on a network of machines.

Consider progs. 1 and 2. Actually, they need not be regarded as distributed programs. The parallel construct is just the most non-deterministic form of sequencing. The parameter mechanism is identical to ALGOL 60 name and value parameter mechanism. And channel and links only restrict the use of shared variables.

Implementation issues, like processor allocation and message buffering, completely disappear from the programmer's concerns. The number of potential parallel activities is independent of the actual number of processors, which can be ignored. The allocation occurs automatically during computation; yet, thanks to the neighbourhood requirement, it is safe and efficient.

ACKNOWLEDGEMENT

Acknowledgement is due to Edsger W. Dijkstra, C.S. Scholten, and the members of the Tuesday Afternoon Club for valuable comments and criticisms.

REFERENCES

- [1] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21,8, August 1978, 666-677.
- [2] E.W. Dijkstra and C.S. Scholten, Termination detection for diffusing computations, EWD 687a, January 1979.